CS1006T Data Strucutres
Unit 1 - Mathematical Background and Intro to DS

Dr. V.A.Kandappan

Assistant Professor,
Department of Computer Science,
Shiv Nadar University Chennai.

Lecture	Tutorial	Practical	Credit
3	0	0	3

SHIV NADAR UNIVERSITY CHENNAI

Syllabus

Prerequisites - CS1001 Programming in C.

1. **Mathematical background and introduction to datastructures** Basic Terminology - Data Organization - Abstract Data Types - Data Structures: Types and Operations - Time and Space Complexity analysis: $\mathcal{O}, \Theta$ and Ω notations - Growth rates - Time-Space trade-off - Time complexity analysis of some example problems. (6 lectures)
2. **List ADT:** Array Implementation of List - Operations on lists: Insertion, Deletion, Merging - Linked Lists: Singly Linked list, Doubly linked list, Circular linked list - Operations on linked lists - The Polynomial ADT - Cursor implementation of lists (5 lectures)

Syllabus (Contd..)

3. **Stack ADT:** Array Implementation, Linked list implementation - Operations on Stacks - Applications of stacks: Balancing Symbols, Postfix expression evaluation, Infix to postfix conversion - Function calls - Recursion. (5 lectures)
4. **Queue ADT:** Array Implementation, Linked list implementation - Operations on Queues - Circular Queue - Double-ended queue - Priority Queue - Applications of Queue. (5 lectures)
5. **Tree ADT:** Implementation of trees - Tree traversals - Binary trees - Binary Search Trees (BST): Operations on BSTs - Expression trees - AVL trees: Operations on AVL trees - Splay trees - Red-Black trees - B-Trees - Heaps - Types of heaps. (6 lectures)

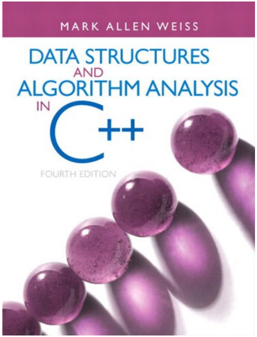
Syllabus (Contd..)

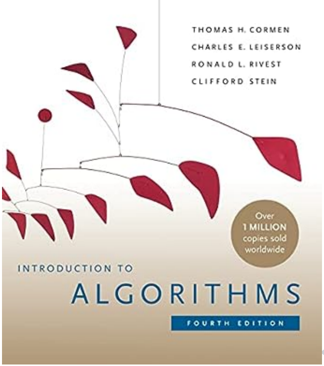
6. **Sorting and Searching:** Searching: Linear Search, Binary Search - Sorting: Bubble sort, Selection sort, Insertion sort, Quick Sort, Merge Sort, Shell sort, Counting Sort. (8 lectures)
7. **Hashing:** Hash Tables - Hash Functions - Separate Chaining - Linear Probing - Quadratic Probing - Open addressing - Rehashing - Extendible hashing. (3 lectures)
8. **Graph ADT:** Implementation of Graphs - Traversal: Breadth First Search, Depth first search - Topological sort (7 lectures)

Total periods: 45

Textbooks and References

1. (CORMEN) Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to algorithms. MIT Press, (2009).
2. (MAW) Weiss, Mark Allen. Data structures and algorithm analysis in C++. Fourth edition, Benjamin/Cummings Publishing Company (2013).





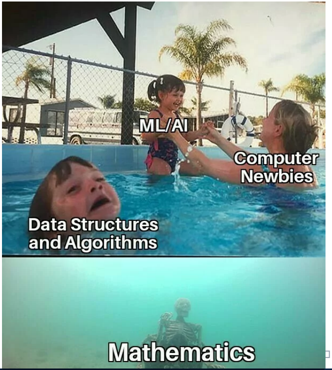
Evaluation Pattern

	Marks
Continuous Assessment	20
Mid Semester	30
End semester	50

Do we need to really need this course?

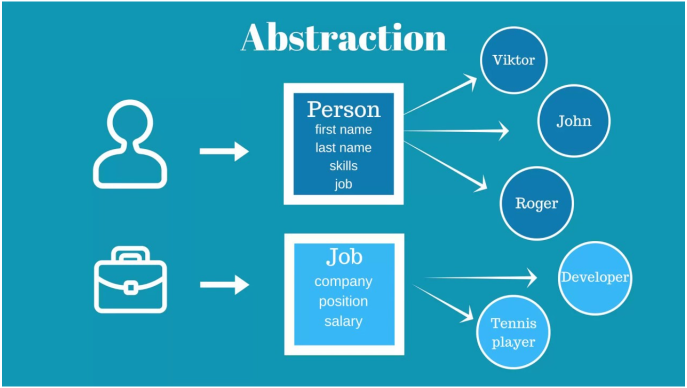
What this course is about?

- How much memory does it take to solve a computational problem in a machine? (**little bit of !!MATH!!** - efficiency of data)
- What is **abstraction** in computer science?
- How do we create, analyse and design custom **data-types**?
- What operations can we do on the custom data structures? (**Operation details** - correctness)



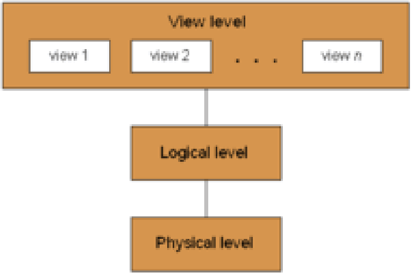
Abstraction in Computer science

"Abstraction is the process of removing unnecessary information so that the computer program runs as efficiently as possible."



Abstraction!

Different levels of Abstraction



Primitive operations in a machine

1. Increment ($a++$, $a--$)/Assignment
2. Compare ($==$, $||$, $\&$, $!=$)
3. Add/Subtract
4. Multiply
5. Modulo Operations
6. Advanced math operations
7. IO operations

Counting primitive operations

```
for(int i=0;i<N;i++)
    do_something();
```

Counting primitive operations

```
for(int  i=0;i<N;i++)
    do_something();
```

do_something(); statement runs N times

Counting primitive operations

```
for(int i=0; i<N; i++)
    for(int j=0; j<N; j++)
        do_something();
```

```
for(int i=0; i<N; i++)
    for(int j=0; j<N; j++)
        do_something();
```

do_something(); statement runs N^2 times

Counting primitive operations

```
for(int i=0; i<N; i++)
    for(int j=i; j<N; j++)
        do_something();
```

Counting primitive operations

```
for(int i=0; i<N; i++)
    for(int j=i; j<N; j++)
        do_something();
```

do_something(); statement runs $\sum_{i=0}^{N}\sum_{j=i}^{N} c$ times

c is the time taken to perform *do_something*(); one time.

Counting primitive operations

```
if(val == true)
    do_something();
else
    do_nothing();
```

Counting primitive operations

```
if(val == true)
    do_something();
else
    do_nothing();
```

do_something(); takes k_1 and *do_nothing*(); takes k_2 then the above code block takes almost $MAX(k_1, k_2)$.

Counting primitive operations

```
if(val < k)
    do_something_1(val);
else if(val > k)
    do_something_2(val);
else
    do_nothing();
```

Counting primitive operations

```
if(val < k)
    do_something_1(val);
else if(val > k)
    do_something_2(val);
else
    do_nothing();
```

do_something(); takes k_1 and *do_nothing*(); takes k_2 then the above code block takes almost $MAX(k_1, k_2)$.

Counting primitive operations

```
void do_nothing(){
    int a = 1, b = 4 , c = 9;
    int disr = b*b - 4*a*c;
    if(disr > 0)
        printf("%d,%d",sqrt(disr), -1*sqrt(disr));
    else
        printf("%di,%dl",sqrt(-1*disr), -1*sqrt(-1
}
```

Count number of primitive operations

Increment/Assignment - 4 Compare - 1 Additions/Sub - 1 Multiplications - 7 sqrt - 4 IO - 2

Counting primitive operations

```
void do_something(){
    a = 100;
    a++;
    while(a<200){
        printf("%d\n",a);
        a += 10;
        b = pow(a,4);
        printf("%d\n",a);
    }
    --a;
}
```

Count number of primitive operations

Counting primitive operations

```
void do_something(){
    a = 100;
    a++;
    while(a<200){
        printf("%d\n",a);
        a += 10;
        b = pow(a,4);
        printf("%d\n",a);
    }
    --a;
}
```

Count number of primitive operations

$3c_1$, $10c_2$,....

Exercise 1

```
for (int i = 0; i < M; i++) {
    for (int j = 0; j < N; j++) {
        c[i][j] = 0;
        for (int k = 0; k < K; k++) {
            c[i][j] += a[i][k] * b[k][j];
        }
    }
}
```

Exercise 2 - Factorial

```
unsigned int factorial(unsigned int n) {
    if (n == 0 || n == 1)
        return 1;
    return n * factorial(n - 1);
}
```

Exercise 3 - Fibonacci Numbers

```
unsigned int fib(unsigned int n) {
    if (n == 0 || n == 1)
        return 1;
    return fib(n - 1) + fib(n-2);
}
```

Exercise 4 - Finding GCD

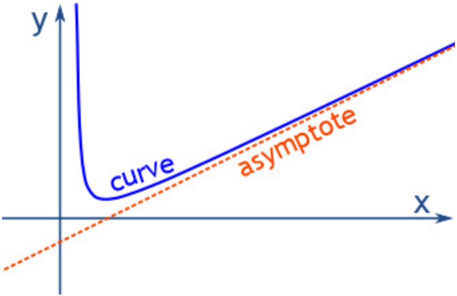
```
long long gcd(long long m, long long n){
    while(n != 0){
        long long rem = m % n;
        m = n;
        n = rem;
    }
    return m;
}
```

$\log_b(N)$

$$\lceil \log_b(N) \rceil$$

Constants are so boring??

what is an Asmptote??



Asymptotic Notations

- $\mathcal{O}\left(f\left(n\right)\right)$
- $\Omega\left(g\left(n\right)\right)$
- $\Theta\left(h\left(n\right)\right)$

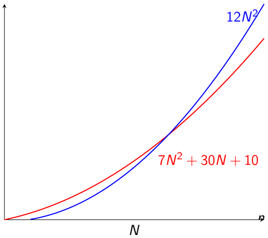
Asymptotic Notations

- $\mathcal{O}(f(n)) \;\not\Longrightarrow\;$ Worst case Analysis (??)
- $\Omega(g(n)) \;\not\Longrightarrow\;$ Best case analysis (??)
- $\Theta(h(n)) \;\not\Longrightarrow\;$ Average case analysis (??)

Big O - $\mathcal{O}(\cdot)$ Definition

$C(n) \in \mathcal{O}(f(n))$ if and only if there exists constants k, n_0 such that

$$C(n) \leq kf(n) \quad \forall n \geq n_0$$



Layman definition

$C(n)$ grows asymptotically no faster than $f(n)$

Big $\mathcal{O}$ notation

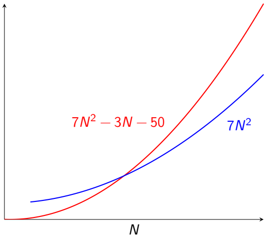
Alternative Big O notation:

$O(1) = O(\text{yeah})$
$O(\log n) = O(\text{nice})$
$O(n) = O(\text{ok})$
$O(n^2) = O(\text{my})$
$O(2^n) = O(\text{no})$
$O(n!) = O(\text{mg!})$

Ω Definition

$C(n) \in \Omega(g(n))$ if and only if there exists constants k, n_0 such that

$$C(n) \geq kg(n) \quad \forall n \geq n_0$$



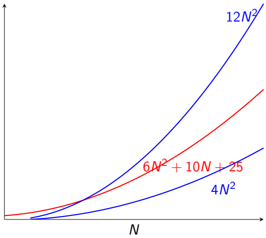
Layman definition

$C(n)$ grows asymptotically no slower than $f(n)$

Θ Definition

$C(n) \in \Theta(h(n))$ if and only if there exists constants k_1, k_2 and n_0 such that

$$k_1 h(n) \leq C(n) \leq k_2 h(n) \quad \forall n \geq n_0$$



Layman definition

$C(n)$ grows asymptotically as fast as $f(n)$

Big $\mathcal{O}$ notation properties

- $C(n) \in \mathcal{O}(f(n))$, then $kC(n)$
- $C_1(n) \in \mathcal{O}(f(n))$, $C_2(n) \in \mathcal{O}(g(n))$, then $C_1(n)C_2(n)$
- $C_1(n) \in \mathcal{O}(f(n))$, $C_2(n) \in \mathcal{O}(g(n))$, then $C_1(n) + C_2(n)$
- $C_1(n) \in \mathcal{O}(C_2(n))$, $C_2(n) \in \mathcal{O}(f(n))$, then $C_1(n) \in \mathcal{O}(f(n))$

Asymptotic notations and bounds

- $C(N) \in \mathcal{O}(f(n))$ - $f(n)$ is the Upper bound for $C(N)$
- $C(N) \in \Omega(g(n))$ - $g(n)$ is the Lower bound for $C(N)$

Asymptotic Analysis - Searching

- Linear Search
  - Worst-case analysis
  - Best-case analysis
  - Average-case analysis
- Binary Search
  - Worst-case analysis
  - Best-case analysis
  - Average-case analysis

Linear Search

Array:

R	U	C	A	T

Search Key: Z

1. Compare Z with R (Not a match)
2. Compare Z with U (Not a match)
3. Compare Z with C (Not a match)
4. Compare Z with A (Not a match)
5. Compare Z with T (Not a match)

Linear Search - Algorithm

Find and return the first occurrence of an element in the array *Arr*.

```
function LINEAR-SEARCH(Arr[N], K)
    flag := −1
    for (i = 0; i < N; i++) do
        if (Arr[i] == K) then
            return i
        end if
    end for
    return flag
end function
```

Linear Search problem

Find and return the first occurrence of an element in the array Arr.

- **Worst-case Analysis** - Worst case occurs if either element is not found or at the last element.
- **Best-case analysis** - Searching for the element that is at the first location.
- **Average-case analysis** - The different run times possible are $1, 2, \cdots, N$. Taking average on all possibilities $\frac{1}{N}\sum_{i=1}^{N} i = \frac{N+1}{2}$ which is $\mathcal{O}(N)$.

Binary Search

Sorted Array:

A	C	R	T	U

Search Key: Z

1. Compare Z with R (Not a match)
2. Compare Z with T (Not a match)
3. Compare Z with U (Not a match)

Binary Search - Algorithm

Find and return the first occurrence of an element in the sorted array *Arr*.

function BINARY-SEARCH($Arr[N], K, start, end$) ▷ Assume $C(N)$
  if $start \leq end$ **then**
    $mid = \left\lfloor \frac{end - start}{2} \right\rfloor$ ▷ takes $\mathcal{O}(1)$
    if $Arr[mid] == K$ **then**
      return mid ▷ takes $\mathcal{O}(1)$
    end if
    if $Arr[mid] > K$ **then**
      BINARY-SEARCH($Arr[N], K, start, mid - 1$) ▷ takes $C\left(\frac{N}{2}\right)$
    else
      BINARY-SEARCH($Arr[N], K, mid + 1, end$) ▷ takes $C\left(\frac{N}{2}\right)$
    end if
  else
    return -1 ▷ takes $\mathcal{O}(1)$
  end if
end function

Binary Search - Asymptotic Analysis

Find and return the first occurrence of an element in the sorted array Arr. The algorithm described has the following recurrence equation:

$$C(N) = C\left(\frac{N}{2}\right) + \mathcal{O}(1) \implies C(N) \in \mathcal{O}(\log(N))$$

- **Worst-case Analysis** - Worst case occurs if either element is not found or at the last element.
- **Best-case analysis** - Searching for the element that is at the first location.
- **Average-case analysis** - The different run times possible are $1, 2, \cdots, \log(N)$. Taking average on all possibilities $\frac{1}{\log(N)} \sum_{i=1}^{\log(N)} i = \frac{\log(N) + 1}{2}$ which is $\mathcal{O}(\log(N))$.

Note on recurrence solution

$$C\left(N\right) = \begin{cases} C\left(\dfrac{N}{2}\right) + \mathcal{O}\left(1\right) & \text{If } N \geq 2 \\ \mathcal{O}\left(1\right) & \text{If } N < 2 \end{cases} \tag{1}$$

Solution to the recurrence (1) using the back substitution method:

Using the above recurrence equation, we know that $C\left(\dfrac{N}{2}\right) = C\left(\dfrac{N}{4}\right) + \mathcal{O}\left(1\right)$ and $C\left(\dfrac{N}{4}\right) = C\left(\dfrac{N}{8}\right) + \mathcal{O}\left(1\right)$

$$C\left(N\right) = C\left(\frac{N}{2^L}\right) + \left(L - 1\right)\mathcal{O}\left(1\right)$$

The above recurrence equation is solved if $\dfrac{N}{2^L} \leq 1$ with which $L \geq \log_2\left(N\right)$.

$$C\left(N\right) = \mathcal{O}\left(1\right) + \left(\log_2\left(N\right) - 1\right)\mathcal{O}\left(1\right) = \log_2\left(N\right)\mathcal{O}\left(1\right) \implies C\left(N\right) \in \mathcal{O}\left(\log_2\left(N\right)\right)$$

Summary on Asymptotic Analysis - Searching

- Linear Search
  - Worst-case analysis - $\mathcal{O}(N)$
  - Best-case analysis $\mathcal{O}(1)$
  - Average-case analysis - $\mathcal{O}(N)$
- Binary Search
  - Worst-case analysis - $\mathcal{O}(\log(N))$
  - Best-case analysis - $\mathcal{O}(1)$
  - Average-case analysis - $\mathcal{O}(\log(N))$

Exercise

Asymptotic bound on finding the maximum element in an array

1. Worst-case analysis
2. Best-case analysis
3. Average-case analysis

Asymptotic bound on finding the maximum element in a presorted array (descending order)

1. Worst-case analysis
2. Best-case analysis
3. Average-case analysis

Back to Finding N^{th} Fibonacci number

Algorithm 1 - A simple recursive function

function REC-FIB(N)
    if $(N == 0 \quad || \quad N == 1)$ **then**
        return 1
    end if
    return REC-FIB($N-1$) + REC-FIB($N-2$)
end function

Complexity Analysis:

- Running time complexity: $\mathcal{O}\left(2^N\right)$
- Memory: $\mathcal{O}\left(1\right)$

Back to Finding N^{th} Fibonacci number

Algorithm 2 - A Fast recursive function

$Arr[0] := 1, Arr[1] := 1, Arr[2:N] = -1$

```
function REC-FIB-FAST(N, Arr)
    if Arr[N-1] != -1 then
        return Arr[N-1]
    else
        Arr[N-1] = REC-FIB-FAST(N-1) + REC-FIB-FAST(N-2)
        return Arr[N-1]
    end if
end function
```

Complexity Analysis:

- Running time complexity: $\mathcal{O}(N)$
- Memory: $\mathcal{O}(N)$

Back to Finding N^{th} Fibonacci number

Algorithm 3 - Fastest non-recursive function

```
function FIB-FASTEST(N)
    if (N == 0  ||  N == 1) then
        return 1
    else
        Fn1 := 1, Fn2 := 1
        for (i = 0; i < N; i++) do
            Ans = Fn1 + Fn2
            Fn2 = Fn1, Fn1 = Ans
        end for
        return Ans
    end if
end function
```

Complexity Analysis:

- Running time complexity: $\mathcal{O}(N)$
- Memory: $\mathcal{O}(1)$

Space-Time Trade-off

Finding N^{th} Fibonacci numbers

Algorithm	Time complexity	Space complexity
Algorithm 1 (Recursive)	$\mathcal{O}\left(2^N\right)$	$\mathcal{O}\left(1\right)$
Algorithm 2 (Recursive-Fast)	$\mathcal{O}\left(N\right)$	$\mathcal{O}\left(N\right)$
Algorithm 3 (Efficient)	$\mathcal{O}\left(N\right)$	$\mathcal{O}\left(1\right)$

"Recursion is not bad; The implementation by programmer (!!you) of the recursion at times is bad."

Types of Space-Time trade-off

- Smaller code vs Loop Unrolling
- Look-ups vs Recalculation
- Compression vs Free data

Types of Space-Time trade-off

- Smaller code vs Loop Unrolling

Smaller Code

```
for(i=0;i<100;i++)
    Arr[i] = 1.0;
```

Loop Unrolling

```
for(i=0;i<100;i=i+2)
{
    Arr[i] = 1.0;
    Arr[i+1] = 1.0;
}
```

Types of Space-Time trade-off

- Look-ups vs Recalculation

Recalculation

```
for(i=0;i<100;i++){
     Arr[i] = sqrt(2)*1.0;
      Arr[i+1] = sqrt(3)*1.0;
}
```

Lookups

```
double a2 = sqrt(2);
double a3 = sqrt(3);
for(i=0;i<100;i++){
     Arr[i] = a2*1.0;
      Arr[i+1] = a3*1.0;}
```

Types of Space-Time trade-off

- Compression vs Free data

Free data

$$A = \begin{bmatrix} 1 & 0 & 6 & 0 & 0 \\ 0 & 6 & 0 & 1 & 0 \\ 0 & 0 & 2 & 0 & 0 \\ 12 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 9 \end{bmatrix}$$

Compression

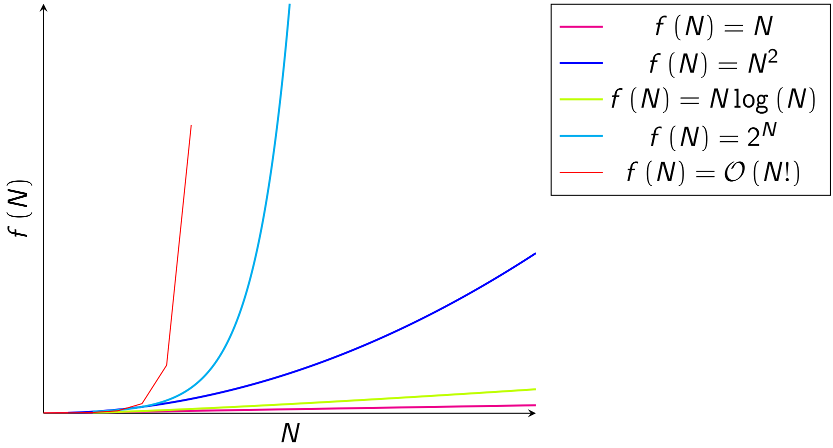
$$D = \begin{bmatrix} 1 & 6 & 2 & 0 & 9 \end{bmatrix}$$

$$N = 5$$

$$id = \begin{bmatrix} 2 & 8 & 15 & 21 \end{bmatrix}$$

$$val = \begin{bmatrix} 6 & 1 & 12 & 1 \end{bmatrix}$$

Growth rates of most common functions



Revisiting meme

Alternative Big O notation:

$O(1) = O(\text{yeah})$
$O(\log n) = O(\text{nice})$
$O(n) = O(\text{ok})$
$O(n^2) = O(\text{my})$
$O(2^n) = O(\text{no})$
$O(n!) = O(\text{mg!})$

Summary of Unit 1

What have we seen till now?

1. Abstraction
2. Time complexity of an algorithm
3. Different asymptotic notations
4. Space-time trade-off
5. Asymptotic analysis